L12 - Testing

February 25, 2020

1 Testing Methods

1.0.1 February 25, 2020

Testing and debugging are closely related.

We have already tested our code manually (in the labs); this is not sufficient for large systems, so we would like to explore *automated* testing. Testing is expensive! Make the computer do it.

1.1 Testing topics

Generally, early in the development of a program, we may have *syntax* errors, which will be spotted by the compiler.

Later in development, we may see *logic errors* where the program runs but returns incorrect results. The compiler cannot help here. Some logical errors have no obvious manifestation.

1.1.1 Test-driven development

Once we write a class, *from the beginning*, we attached a test class to it.

Ideally we would have a different person develop the tests than wrote the code under test. In large companies these may be entirely different job descriptions.

Error prevention

- use good software development practices, like encapsulation
- try to decouple your classes as much as possible to avoid weirdness in the interfaces
- *cohesion*: make the same thing happen in the same object
- use modularization to make errors more obvious
- write good documentation to make errors detectable

1.2 Testing vs. Debugging

Testing is the practice of searching a class or method for errors. Debugging is about finding the exact source of an error - which might be some distance from where it manifests - and then fixing the error.

1.2.1 Test Techniques

Unit Testing Each *unit* of an application can be tested - like a class or module. Do this during development - fixing errors early saves time and money. Fixing errors after deployment is especially expensive.

First, understand what the unit should do - its *contract* with the outside world.

Test the *boundary conditions*. If you have an array:

- test a full array
- test an empty array
- add to a fully array
- remove from an empty array
- remove the last element
- and more…?

This and other checks will make up the *test suite*.

Regression Testing After making changes to the code, you should run your test suite again in order to make sure you have not introduced an exciting new bug. This practice is called *regression testing*.

Test Automation Good testing is important, but throrough testing is time consuming and repetitive. Good thing we have a computer! Using a *test rig* or *test harness* relieves some of the burden.

An additional test harness is written to automate testing. Objects of the harness replace typing stuff in manually. You need to keep the test classes up to date as you add functionality to the class under test.

Test frameworks like `jUnit` exist to support automation. Look at the *online-shop-junit* project.

Assertions Used to compare the results of the unit under test against the desired result.

`setUp()` and `tearDown()` Defined in every `jUnit` test class. `setUp()` runs before the test suite and `tearDown()` runs after.

1.2.2 Unit Tests in BlueJ

BlueJ's interface allows you to

- create class instances
- invoke individual class methods
- inspectors provide a view into an object's state

Doing the thing

- select View → Show Team and Test Controls, then click Run Tests
- right click the test class in the boxes/arrows pane, and then select a test to run it

1.3 Debugging Methods

1.3.1 Manual walkthroughs

Because coding exams are *so much fun*. Low tech, often underappreciated.

1.3.2 Tabulating Object State

Read the code, and follow the state of objects through program execution

1.3.3 Verbal walkthroughs

Explain your problem to someone else. They might find it or you might because you explained it.

1.3.4 Print Statements

Super common - just print useful chunks of state to the terminal. Works in every language.

[ ]: